Java Qsns Assignmnt

Date 18 Apr 2024

Lt Col T Varadharajaperumal(Retd)

Qsn 1. What is Assertion in Java. Give examples?

Ans 1. It is a statement that is used to express assumptions or expectations of the program at a particular point of execution. If an Assertion fails, it means the condition being asserted has failed or is false , and hence an assertion error is thrown , meaning there is an error in the code at that point.

Assertions can be used to catch bugs during development by checking that the program is in a valid state at different points in execution. They are not intended for error handling in production code; instead, they are used as a debugging aid during development and testing.

Example : -

public void processNumber(int number) {

assert number > 0 : "Number must be positive"; // Assertion that checks if number is positive

// Code to process the number

}

In this example, the assertion checks if the number argument is greater than zero. If the condition is false, an ‘AssertionError’ is thrown with the message "Number must be positive."

Qsn 2. Differentiate between Error and Exception?

Ans 2. In Java, both Error and Exception are classes that represent different types of problems that can occur during the execution of a program. However, they are used to handle very different scenarios:

Error:

Error is a class that represents serious issues that are usually beyond the control of the programmer and should not be caught or handled in typical application code.

Errors are generally related to issues in the environment where the program is running, such as a lack of memory (OutOfMemoryError) or a failure in the Java Virtual Machine (JVM), such as InternalError.

Errors are uncommon and typically indicate a serious failure in the system that the program cannot reasonably recover from. Therefore, they should not be handled in application code.

Examples of Error include:

OutOfMemoryError: Thrown when the JVM runs out of memory.

StackOverflowError: Thrown when a thread's stack overflows, typically due to excessive or infinite recursion.

Exception:

Exception is a class that represents issues that can be handled and recovered from in the application code.

Exceptions are typically related to problems within the control of the programmer, such as invalid input, I/O errors, or resource access issues.

Exceptions are divided into two categories:

Checked exceptions: These exceptions must be either caught or declared in the method signature using the throws keyword. They represent expected problems that the program should handle (e.g., IOException).

Unchecked exceptions (also known as runtime exceptions): These exceptions do not need to be declared or caught. They represent unexpected issues such as programming errors (e.g., NullPointerException, ArrayIndexOutOfBoundsException).

Examples of Exception include:

IOException: Represents I/O-related issues such as failure to open a file.

NullPointerException: Represents accessing an object that is null.

In summary, Error is used to represent serious system-level problems that are usually beyond the programmer's control and should not be handled in application code, while Exception is used to represent problems that can and should be handled in application code.

Qsn 3. What is Daemon threads in Java.

Ans 3.

In Java, a daemon thread is a special type of thread that runs in the background and is designed to support other threads in the application. Daemon threads do not prevent the program from exiting; when all non-daemon threads have finished executing, the program will terminate even if daemon threads are still running.

Here are some key points about daemon threads in Java:

Definition: A daemon thread is a thread that runs in the background and provides support for other threads in the application.

Purpose: Daemon threads are often used for tasks such as garbage collection, logging, and monitoring. They provide essential background services that keep the application running smoothly.

Life Cycle: The JVM terminates when all non-daemon threads have completed their execution. Daemon threads do not keep the JVM alive; once all user (non-daemon) threads have finished, the JVM will exit, and any remaining daemon threads will be stopped.

Creating a Daemon Thread: To create a daemon thread, you need to set the thread as a daemon before starting it. This can be done using the setDaemon(true) method on a Thread object:

Thread daemonThread = new Thread(() -> {

// Thread code goes here

});

daemonThread.setDaemon(true); // Set the thread as a daemon thread

daemonThread.start(); // Start the thread

Differences from Non-Daemon Threads: Unlike non-daemon threads, daemon threads do not block the JVM from exiting when the program has finished executing. They are often used for tasks that do not need to be completed before the program ends.

Common Uses: Daemon threads are commonly used for background tasks such as periodic cleanups, monitoring, and other low-priority tasks.

In summary, daemon threads in Java are background threads that support the main application threads and do not prevent the JVM from exiting when the program completes.

Qsn 4. Give any 5 built-in-exception for Checked and Unchecked exception?

Ans 4.

In Java, exceptions are categorized into two main types: checked exceptions and unchecked exceptions.

Checked Exceptions:

Definition: Checked exceptions are exceptions that are checked at compile time. This means that the compiler will enforce the handling of these exceptions.

Handling: When a method throws a checked exception, it must be either caught using a try-catch block or declared in the method signature using the throws keyword.

Examples: Examples of checked exceptions include IOException, SQLException, ClassNotFoundException, and FileNotFoundException.

Purpose: Checked exceptions are used to handle recoverable issues, such as file I/O problems or database connection issues. They represent expected problems that can be anticipated and handled by the program.

Unchecked Exceptions:

Definition: Unchecked exceptions, also known as runtime exceptions, are exceptions that are not checked at compile time. The compiler does not require handling of these exceptions.

Handling: Unchecked exceptions do not need to be caught or declared in the method signature. However, they can still be caught using a try-catch block if desired.

Examples: Examples of unchecked exceptions include NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException, IllegalArgumentException, and ClassCastException.

Purpose: Unchecked exceptions are generally caused by programming errors or unexpected conditions in the code. They often indicate a bug or flaw in the logic of the program.

Differences:

Compile-Time Checking: Checked exceptions must be either caught or declared in the method signature, while unchecked exceptions do not need to be explicitly handled.

Purpose: Checked exceptions are used for recoverable issues and expected problems, while unchecked exceptions are often indicative of programming errors or unexpected issues.

Handling: Checked exceptions require explicit handling in the code, either through a try-catch block or by declaring them in the method signature. Unchecked exceptions can be caught if necessary, but it's not mandatory.

Impact: Since checked exceptions are explicitly handled, they provide a way to deal with specific failure scenarios in a controlled manner. Unchecked exceptions, on the other hand, may cause unexpected crashes if not addressed properly.

Top of Form

Here are 5 examples of each:

Checked Exceptions:

IOException: Represents I/O-related issues such as failure to open a file, read/write errors, and more.

SQLException: Represents issues that occur during database operations, such as connectivity problems or query execution errors.

ClassNotFoundException: Thrown when an application tries to load a class that cannot be found in the classpath.

FileNotFoundException: Thrown when an attempt to open a file specified by a path fails because the file does not exist.

InterruptedException: Thrown when a thread is interrupted while waiting or sleeping.

Unchecked Exceptions:

NullPointerException: Thrown when trying to access an object reference that is null.

ArrayIndexOutOfBoundsException: Thrown when trying to access an array element using an invalid index (e.g., a negative index or an index beyond the length of the array).

ArithmeticException: Thrown when an arithmetic error occurs, such as dividing by zero.

IllegalArgumentException: Thrown when an illegal argument is passed to a method, typically due to invalid input.

ClassCastException: Thrown when attempting to cast an object to a type to which it cannot be cast.

These are just a few examples of checked and unchecked exceptions in Java. Checked exceptions are designed to be caught and handled by the developer, while unchecked exceptions are generally caused by programming errors and are not usually caught in the application code.

Qsn What is the difference between Scanner class and Data input stream class?

In Java, the Scanner class and the DataInputStream class are both used for reading data, but they serve different purposes and offer different levels of functionality and usability.

Scanner Class:

**Purpose**: The Scanner class is a versatile and user-friendly class used for parsing and reading text input, often from user input, files, or other input streams.

**Ease of Use**: Scanner is designed to be easy to use, providing convenient methods for reading primitive data types (e.g., int, double) and strings from a variety of sources such as the console (System.in), files, and strings.

**Parsing**: Scanner supports methods for tokenizing and parsing input based on delimiters (default is whitespace). It provides methods such as next(), nextInt(), and nextDouble() for reading and parsing data.

**Buffering**: Scanner uses an underlying buffer to read data efficiently, allowing it to handle various input sources.

**Flexibility**: Scanner is flexible and can be used for interactive applications, file parsing, and reading formatted data from various sources.

DataInputStream Class:

**Purpose**: The DataInputStream class is used for reading primitive data types (e.g., int, double) and strings from an input stream in a binary format.

**Binary Data**: Unlike Scanner, which is designed for parsing text input, DataInputStream is specifically for reading binary data in a specific format (e.g., little-endian or big-endian).

**Methods**: DataInputStream provides methods such as readInt(), readDouble(), readUTF(), and others for reading data types directly from the input stream.

**Stream-Based**: DataInputStream works directly with input streams, such as FileInputStream or ByteArrayInputStream, and reads data in a continuous stream.

**No Tokenization**: Unlike Scanner, DataInputStream does not provide tokenization or parsing capabilities based on delimiters; it reads data directly from the stream in a binary format.

Key Differences:

**Purpose**: Scanner is designed for parsing and reading text input, while DataInputStream is designed for reading binary data.

**Data Format**: Scanner reads text input and provides parsing capabilities, while DataInputStream reads data directly from a stream in binary format.

**Use Case**: Scanner is ideal for interactive applications, file parsing, and reading formatted text data, while DataInputStream is best suited for reading binary data from files, network streams, and other sources.

Both classes have their own strengths and are suitable for different types of applications, so the choice between them depends on the specific requirements of your project.

Top of Form

**Qsn 6. What is Serialization, why do we need it?**

**Serialization** is the process of converting an object's state into a byte stream so that it can be saved to a file, sent over a network, or stored in a database. This byte stream can later be deserialized to recreate the object in memory with the same state.

Why Do We Need Serialization?

**Persistence**: Serialization allows you to save the state of an object to a file or database so that it can be restored later. This is useful for saving user data, application state, or other important information.

**Communication**: When sending data over a network (e.g., in distributed systems or web services), objects often need to be serialized into a byte stream for transmission. The receiving end can deserialize the byte stream to recreate the object.

**Caching**: Serialization can be used to cache objects for quick retrieval later. For example, caching complex data structures can improve performance in applications.

**Deep Copy**: Serialization can be used to create a deep copy of an object. By serializing an object and then deserializing it, you can create a new instance with the same state.

**Interoperability**: Serialization allows you to convert objects to a standard format that can be understood by different systems or programming languages, facilitating data exchange and interoperability.

How Serialization Works in Java:

In Java, serialization is achieved using the java.io.Serializable interface:

Serializable **Interface**: To enable serialization for a class, the class must implement the Serializable interface. This is a marker interface, so it does not define any methods.

**Serialization Process**: The ObjectOutputStream class is used to serialize objects. It writes the object's state to a byte stream.

**Deserialization Process**: The ObjectInputStream class is used to deserialize objects. It reads the object's state from a byte stream and recreates the object in memory.

Here's a basic example of serialization and deserialization in Java:

java  
import java.io.\*;

class Person implements Serializable {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public String toString() {

return "Person{name='" + name + "', age=" + age + '}';

}

}

public class SerializationExample {

public static void main(String[] args) {

Person person = new Person("John", 30);

// Serialization

try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("person.ser"))) {

oos.writeObject(person);

System.out.println("Object serialized.");

} catch (IOException e) {

e.printStackTrace();

}

// Deserialization

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("person.ser"))) {

Person deserializedPerson = (Person) ois.readObject();

System.out.println("Object deserialized: " + deserializedPerson);

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}